A Distributed Data Mining Framework Accelerated with Graphics Processing Units

Nam-Luc Tran, Quentin Dugauthier and Sabri Skhiri
Euranova R&D
Belgium
Email: {namluc.tran, quentin.dugauthier, sabri.skhiri}@euranova.eu

***Abstract*—In the context of processing high volumes of data, the recent developments have led to numerous models and frameworks of distributed processing running on clusters of commodity hardware. On the other side, the Graphics Processing Unit (GPU) has seen much enthusiastic development as a device for general-purpose intensive parallel computation. In this paper we propose a framework which combines both approaches and evaluates the relevance of having nodes in a distributed processing cluster that make use of GPU units for further fine-grained parallel processing. We have engineered parallel and distributed versions of two data mining problems, the naive Bayes classifier and the k-means clustering algorithm, to run on the framework and have evaluated the performance gain. Finally, we also discuss the requirements and perspectives of integrating GPUs in a distributed processing cluster, introducing a fully distributed heterogeneous computing cluster.**

I. Introduction

Data mining and machine learning keep on gaining importance in our society. The data generated by our interactions is used to conduct business decisions or influence our interactions with services. Take for example the "Recommended Products" section of an online shopping website or the "People you may know" recommendation of a social network. All these features consume high volumes of data and require fast computations in order to provide the user with relevant information and accurate recommendations as soon as fresh information about the user is available.

The past years have seen the development of general-purpose computing on Graphics Processing Units (GPU). At a similar cost, a GPU can theoretically, as of today, deliver more parallel computing performance than a traditional CPU.

However, programming with GPUs often requires to delve into specific device code (CUDA, OpenCL) and low-level hardware considerations such as memory allocation, bank selection and memory access patterns. Many libraries however, such as Thrust[1], present a higher-level abstraction of the device-specific concepts and provide primitives for programming on the GPU. These implement the best practices and programming patterns in order to use the GPU in general purpose computing. There also exist libraries which are specifically developed for problems in the fields of signal processing, image processing or data mining. All these existing libraries provide solutions for writing code running on a machine equipped with one or more GPUs and are focused on the complexity of the algorithms rather than the volume of data being processed.

[1]Thrust - http://thrust.github.io/

In parallel to this, the recent advances in Cloud Computing and Big Data processing have led to the development of distributed and parallel programming models, as Google's MapReduce[1], and other frameworks based on the more general DataFlow Graph model, such as Dryad[2], Nephele[3] and AROM[4]. These frameworks allow to launch distributed parallel data processing jobs on high volumes of data. Numerous reports have already evaluated the performance and scalability of the programming models supported by many of these frameworks [5], [6].

In this work we propose a framework suited for data mining combining the distributed execution model of the aforementioned frameworks with the power of the GPUs available on the individual computing nodes. This adds a second level of parallelization as the job can be scheduled in parallel on the nodes of the cluster, and each node will execute its routine using parallel primitives of its equipped GPU. As a means to demonstrate the framework and the execution model we have developed a version of the naive Bayes classifier and the k-Means clustering algorithm which both make use of the distributed cluster setup and the massive parallel computation power provided by the GPU on the nodes. To our knowledge no existing framework has so far combined the distributed processing approach with the parallel processing power of individual nodes.

The rest of this paper is organized as follows: Section 2 presents the related work and the motivations behind the project, Section 3 presents the architecture of the framework, Section 4 details the algorithms that run on the framework, Section 5 and 6 discuss the test results on the two implemented use cases, Section 7 outlines our insights about the integration of our framework model to existing DFG processing models and, finally, Section 8 concludes our work.

II. Related Work

Paralellizing and distributing the implementation of the data mining techniques has proven to be an effective solution when dealing with high volumes of data. The main principle is to speed up the data mining operations by adding more machines to the cluster rather than modifying the algorithm with complex and finely selected optimizations.

Many of the classical data mining algorithms can be parallelized and distributed by building a local model on each node and then combining these models together to form the global model. [7] has performed a survey on the parallelization and the complexity of some of the most used data mining

algorithms, including k-NN, decision trees, naive Bayes, k-means, PageRank, SVM, LDA and CRF.

This concept can be pushed further and [8] generalizes the principle as "*summation form*". This refers to the form of the algorithm where the data is divided into as many pieces as there are cores, and each core processes the equation over its share of data. The individual results are finally aggregated at the end to form the global result. Many data mining algorithms fall under this model and can be implemented with the MapReduce paradigm.

Recently, the same kind of concept has also been applied to the distributed training of the structured perceptron in which each perceptron individually trains on a disjoint subset of the data and eventually the final parameters of the model are set to a weighted mixture of the parameters of each model [9].

The GPU on the other hand has also proven to be relevant for data mining operations such as k-means clustering, Monte Carlo simulations, similarity joins and itemset mining. Numerous studies have shown significant speedups of these operations over their CPU implementations ([10], [11], [12], [13], [14]). There also exist libraries specifically designed to solve particular problems in numerical simulation of fluids, signal and image processing. In data mining, projects such as [15] and [14] have developed libraries which implement optimized machine learning algorithms for running on a single node where at least one GPU is available.

Most of the existing studies on data mining on GPU only focus however on the speedup gained from the computation on GPU compared to the CPU implementation, sometimes with multiple GPUs equipped on the same machine. None so far has explored the setup where distributed nodes in a cluster are each equipped with a GPU compute capable device.

This particular setup presents two levels of parallelism provided first by the distributed nature of the cluster and secondly by the parallel processing abilities of the GPU device equipped on the nodes. Both levels of parallelism are distinct and require specific engineering of the algorithm to run distributed and to harness the individual parallel processing power on the nodes. Furthermore, special care should even be addressed when developing an algorithm for the GPU from an existing sequential design as a bad parallel implementation can lead to worse performance. Finally, this setup also raises additional challenges on the scheduling as it also has to take into account the individual compute capabilities of the nodes.

III. Architecture of the Framework

In this study we propose a framework that enables the execution of data processing tasks in a distributed setup and that takes into account the compute capabilities of the GPU hardware that is available on each node.

The framework is suited for running data processing algorithms composed of the following phases:

1) Independent computation of a partial model on disjoint data chunks.
2) Combination of the partial models into the global model.

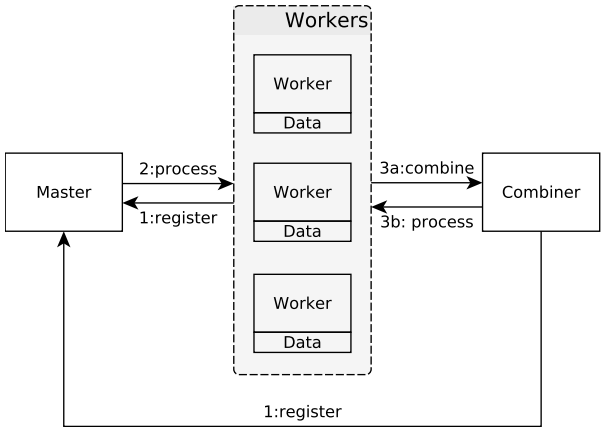

Fig. 1. Architecture of the framework. Each worker node is equipped with a compute-capable GPU. Steps 3 *a* and b can be repeated in the case of an iterative algorithm.

3) Iterate again if needed, until a convergence criteria is met.

The whole dataset is first divided into exclusive data chunks. Each one will feed an individual worker in the cluster. Figure 1 depicts the overall architecture of the framework.

The individual worker nodes are each equipped with a compute-capable GPU. Each node runtime can be run independently on a machine. At the first step all the *worker* nodes register to the *master* node. They each receive instructions about their respective role, as well as all the job information and cluster configuration. The worker nodes will get information from the master about the data chunks they will work on and the location of the subsequent *combiner* node.

We consider that the data is stored in a distributed fashion over the worker nodes so that each worker only needs to load and work on data that is already locally present and does not require a transfer of data through the network. This can be achieved by using a distributed filesystem with a sharding policy such as HDFS[2].

This architecture is much similar to the MapReduce model (each individual worker can be seen as executing a map task and the combiner performs a reduce task over the outputs of the map tasks) with the exception that the workers are not limited to run only map tasks but can execute any complex task (even potentially a full MapReduce or DFG job, as will be discussed later). The framework provides the signalling mechanism for the communication between the nodes and the different roles.

IV. Algorithms

In order to demonstrate the framework we have engineered two classical machine learning algorithms to run on the framework: the naive Bayes classifier and the k-means clustering algorithm. In this section we describe these algorithms and discuss the required considerations when developing algorithms for this framework.

The reengineering covers two phases: first, the algorithm needs to be data-parallel, consisting in a phase where the same

[2]Apache Hadoop FileSystem http://hadoop.apache.org/

algorithm is applied on different nodes on different chunks of data, with the individual results merged at the end. Secondly, the data-parallel phase of the algorithm running on the workers needs to be designed in a way such that the parallel computing power on the GPU is harnessed.

A. Naive Bayes Classifier

The naive Bayes classifier is an important supervised classification method based on the Bayes' theorem and which assumes that the presence of a feature in a class is independent of any other feature. The method however also shows good performance even in the case of strong dependencies [7].

Let T the training set of data containing n instances in k classes, each instance is represented as a feature vector $(x_1, \ldots, x_d)$ associated with a class label v. The general case of the naive Bayes classifier is described as follows:

1) Estimate $P(y = v)$ from the proportion of instances of class v in T.
2) Estimate $P(x_i = u_i | y = v)$.

In the case x_i is a continuous variable, a common strategy is to assume that x_i follows a Gaussian distribution. Otherwise, if x_i is categorical then the estimation can be done by counting the fraction of records where $y = v$ and $x_i = u$

Given a new object $(x_1, \ldots, x_d)$, the prediction of its class y is given by the following computation:

$$y = \underset{v}{\operatorname{argmax}} P(y = v | x_1 = u_1, \ldots, x_d = u_d)$$

$$y = \underset{v}{\operatorname{argmax}} \prod_{i=1}^{d} P(x_i = u_i | y = v) P(y = v)$$

The parallelization of the naive Bayes classifier is straightforward. Given a cluster of c nodes, the T set can be partitioned into c distinct shards, each assigned to a node. Each node computes the distribution of $P(x = u | y = v)$ on its training instances. Assuming Gaussian distributions, each node in practice computes the mean and the variance of each feature for each class in the case of continuous variables. The means and variances computation operation is highly parallelizable and can make use of the GPU compute capabilities on the node.

The means and variances are computed individually by each node using a parallel summation reduction and transformation both performed highly in parallel by the GPU on the workers. These then need to be combined from all the nodes to provide the means and variances of the distributions on the whole dataset. We use the combined variance method to sum up the partial means and variances results:

$$m = \sum_{j=1}^{g} \frac{k_j m_j}{n}$$

$$v = \frac{1}{n-1} (\sum_{j=1}^{g} (k_j - 1) v_j + \sum_{j=1}^{g} k_j (m_j - m^2))$$

where m is the mean, v the variance, g the number of partitions, k_j the size of the j^{th} shard, and $n = \sum k_j$ the total number of data.

Algorithms 1 and 2 summarize the routines executed on the workers and the combiner.

Algorithm 1 Distributed Naive Bayesian Classifier - Worker Routine

```
Read data from local data partition in GPU memory T
for all classes v_i in T do
    for all features f_j do
        M[i][j] ← Compute Mean on GPU
        V[i][j] ← Compute Variance on GPU
    end for
end for
Send M and V to Combiner
```

Algorithm 2 Distributed Naive Bayesian Classifier - Combiner Routine

```
Receive model parts M_w and V_w from all the w workers
for all classes v_i do
    for all features f_j do
        M[i][j] ← Combine means from M_w
        V[i][j] ← Combine variances from V_w
    end for
end for
```

B. k-Means Clustering

The k-means clustering algorithm is a cluster analysis method which partitions n observations into k clusters with each observation belonging to the cluster with the nearest mean. Concretely, given a set T of n data points in R^d such as $X_i = (x_1, \ldots, x_d)$, the problem is to find k points M_j in R^d such that the following measure is minimized:

$$\frac{1}{n} \sum_{i=1}^{n} (\min_j d^2(X_i, M_j))$$

where $d(X_i, M_j)$ is the distance (usually the Euclidian distance) between X_i and M_j. The k points $\{m_j\}$ are known as the *cluster centroids*. In short, the problem can be summarized as finding k cluster centroids such that the average squared (Euclidian) distance (or Mean Squared Error - MSE) between a data point and its nearest centroid is minimized.

We base our implementation of the k-means algorithm on the distributed version proposed by [16] in which the initial data set is partitioned among all the working nodes. Each node then receives the same set of initial centroids and performs the k-means iterations on its share of data. At the end of each iteration, the updated centroids positions and the number of data points belonging to each cluster are gathered by the combiner in order to globally update the positions of the centroids.

The iterations performed by the k-means algorithm are all data-intensive operations which can be congruently parallelized on the GPU device. At each iteration the following steps are performed:

1) Compute the distance of each data point to each of the k centroids.
2) For each data point, determine its closest centroid, and keep a count of the number of data points related to each centroid.

3) Update each centroid with the coordinates from its related data points.

Each of the aforementioned steps can be designed as a parallel operation to be run on the GPU. The first step can be implemented as a parallel transformation followed by a summation reduction of all the data points for each centroid in the same way as step 3. Using the row-major representation of 2D data, we are able to make use of the performant access patterns on GPUs such as coalesced accesses and perform step 2 without the need to perform a costly matrix permutation. Algorithms 3 and 4 summarize the steps taken by the worker and the combiner.

Algorithm 3 Distributed k-means - Worker Routine

```
Read data T from local data partition in GPU memory
Set active = true
while active do
    centroids ←Receive k centroids {m_j} from combiner
    for all centroids m_j do
        D_j ← Compute all d²(x_i, m_j)
    end for
    idx ← Identify index j of centroid closest to x_i
    ct' ← Increment count number of matched centroid
    MSE' ← Sum all d²(x_i, m_j) for x_i to its closest m_j
    for all m_j do
        {m'_j} ← Contribute each x_i coordinates to its matching m_j
    end for
    Send {m'_j}, ct' and MSE' to Combiner
    active ← get active response from Combiner
end while
```

Algorithm 4 Distributed k-means - Combiner Routine

```
Generate k initial centroids {m_j}
MSE ← Large number
while MSE > threshold do
    Send {m_j} and active to the w workers
    Receive model parts {m_j}', ct' and MSE' from all w workers
    ct ← Σ ct'
    m_j ← (Σ m'_j) / ct_j
    MSE ← Σ MSE'
end while
Send active ← false to the w workers
```

C. General Case

In the two previous use cases we have taken two data-parallel algorithms and have engineered them to run on our framework. We will discuss here the general requirements of any algorithm that will run on the framework in order to optimally exploit this processing model.

1) Data-parallelism: The framework offers two layers of parallelism: the first layer is at the worker level with the distributed nodes in the cluster and the second is at the processing level on each worker thanks to the parallel compute capabilities of GPU devices. In order to fit the framework, the algorithm should be data-parallelizable. This basically means that the outcome of the algorithm should be identical whether it is applied to a single block of data or on disjoint chunks of the data then combined back. Algorithms that fit the MapReduce or DFG model as discussed in Section 2 fall under this requirement.

In our framework each worker has to compute a local model based on the same algorithm but applied to its local data chunk. In the naive Bayes classifier use case, each worker computes the means and variances of the features and the probability of each class on its own dataset. In the k-means use case, each worker performs a full k-means iteration based on the data points in its local set.

2) GPU parallel processing: The previous step then needs further engineering as these local computations will make use of the GPU compute devices. By experience this step requires a full rethinking of the way the algorithm works in order to design an algorithm which benefits from the power of parallel computing on GPU, taking into account the details and characteristics of GPU programming. A straight transposition of the algorithm to run on the GPU will often lead to very poor and even worse performance. This is the phase that requires the most engineering, at least at the time being with the current state of the GPU development tools.

3) Model outcome: The final step is to identify how the final outcome of the model can be obtained as the combination of each of the partial model produced by the individual workers.

In the naive Bayes classifier use case, the outcome of the model consists of means and variance values. These are combined together from the individual workers partial models thanks to the combined means and variances method. In the k-means use case, the intermediate counts of the number of data points belonging to a centroid are saved in order to compute the final position of the centroids based on the intermediate results of the workers.

V. Test and Results

The framework has been tested against the distributed naive Bayes classifier and the distributed *k*-means clustering algorithms detailed in the previous section. Both use cases have been tested with the KDD Cup 1999 Data Set[3] with various levels of Gaussian noise added in order to inflate the total volume of data from the initial 0.6Gb to 7.2Gb.

The dataset contains raw TCP connection information over a U.S. Air Force LAN. Each connection presents 41 features (32 continuous + 9 discrete) and is labeled as normal or as a network attack, with exactly one attack type, for a total of 23 classes. The data mining task is to build a model for the classification of connections into these classes.

Note that we are here interested in measuring the performance of the computation of the global model, which purpose is to illustrate the use cases of the framework, rather than the performance and precision of the constructed models.

All tests have been performed on 4 physical machines equipped with Intel i5 3550, 16Gb RAM and Gigabit interconnect. The GPUs used for testing include the Nvidia GTX

[3] http://archive.ics.uci.edu/ml/datasets/KDD+Cup+1999+Data

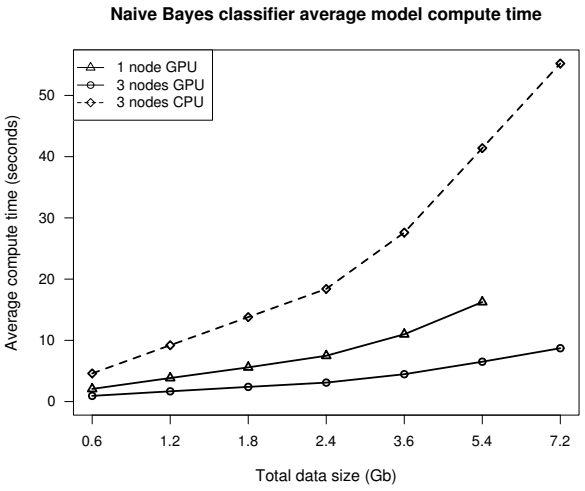

Fig. 2. Average time spent for the computation of the model on a GPU node (Titan) and on a CPU node for various sizes of the dataset. Each of the three workers in the cluster locally processes one third of the total data size.

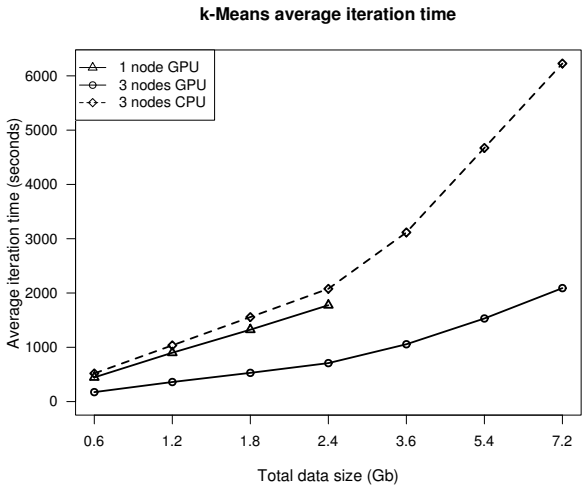

Fig. 3. Average time spent for the computation of a k-means iteration on a GPU node (Titan) and on a CPU node for various sizes of the dataset. Each of the three workers in the cluster locally processes one third of the total data size.

Titan (6Gb device memory) and the Nvidia Tesla K20 (5Gb device memory). The master and the combiner run on the same host without GPU.

For each case, we test the performance of the GPU implementation on a single node, then on a distributed setup with three nodes and for various sizes of the dataset. We finally compare these results with the CPU implementation distributed on three nodes. We also have conducted the tests of the CPU implementation on a single node and the results are constantly slower than the results for the test distributed on 3 nodes roughly by a factor of 3. We have omitted these results on the plots for better readability.

A. Distributed Naive Bayes Classifier

For the computation of the naive Bayes classifier each worker locally loads an exclusive subset of the whole dataset. Each node then computes a naive Bayes classifier on its subset of data. The global model is then derived on the combiner using the combined means and variance method as described in Section 4. The computation of the local model makes use of the GPU parallel reduction algorithms.

Figure 2 represents the average time spent measured on one worker (equipped with GTX Titan) for the computation of the model in function of the dataset size, each of the three workers locally processing one third of the total data size. The tests on the single-node setup have not been performed on all the dataset sizes as, in this implementation of the algorithm, the entire dataset needs to fit in the memory of the GPU.

First we can observe that the GPU implementation of the job takes less time to complete than its CPU counterpart. The distributed GPU setup in this case offers a speedup of up to 7 compared to the computation performed distributed over three CPU workers for the largest dataset size. The speedup increases with the size of the data as the overhead of setting up the GPU gets smaller with regards to the time spent on the actual processing.

We can also observe a linear scalability on the amount of distributed nodes and on the size of the data for the GPU implementation. The GPU implementation of the naive Bayes classifier is thus able to linearly scale with the size of the dataset.

B. Distributed k-means

In the distributed k-means use case, each worker individually loads an exclusive chunk of the complete dataset, then receives the set of initial centroids and performs a k-means iteration on it. At the end of each iteration the MSE and the updated centroids are merged by the combiner. Globally updated centroids are then broadcasted in return by the combiner for the next iteration if needed, depending on the global value of the MSE. All the data sizes could not have been tested on the single-node setup as the whole dataset partition needs to reside inside the device memory in this implementation.

We have run the distributed k-means algorithm on the dataset over 32 numerical features of the data and a k value of 23 (number of initial clusters). Figure 3 shows the mean iteration performance on a node (equipped with GTX Titan) of the framework on different sizes of data. Each of the three workers locally processes one third of the total data size. We can observe a time speedup on average of 3.5 for the GPU version over the CPU version. For the dataset sizes which can be run on a single-node, we can observe that the performance is close to the three-nodes setup that only makes use of the CPU.

VI. Results Discussion

It seems important to us to stress out at this point that all the performance comparison figures concerning the computation on CPU should be taken for illustration only. It would be irrelevant to compare these figures as an apples-to-apples comparison of the performance of GPU vs CPU for two reasons. The first reason is that the CPU and the GPU used in

the tests do not compete in the same category of the market. To provide a fair comparison one should use hardware similarly priced and targeting the same end of the market. Then, in order to perform such comparison, the CPU implementation should be finely engineered to make use of all the lower level specialized instruction sets, such as, in this case, the SIMD extensions (SSE) as explained in [17].

Instead, these figures should be considered with the context of distributed processing frameworks as background. What we want to highlight is the performance gain when, for a same job, and given a cluster of distributed processing nodes, some nodes are equipped with a GPU and run a GPU-optimized implementation of the job. And within this scope the results show a considerable performance gain of the compute nodes when they make use of the parallel processing power of the GPU for massive data-parallel tasks. This adds a second level of speedup and parallelization to the already distributed nature of the compute cluster.

Concerning the data transfer, which is often considered an inconvenient when working with GPUs, we have observed that under the right conditions e.g. when using memory representations that fits the GPU programming models such as row or column-major representations, the majority of the PCIe 2.0 and 3.0 bandwidth (8 Gb/s and 12 Gb/s) between the RAM memory and the GPU device can be leveraged in memory transfers. Indeed, in our tests, the whole data set is first entirely copied to RAM memory then moved to the GPU device memory. The timings of these transfers are significantly low and represent less than 1 percent of the total processing time. In our measures, filling the 6Gb of memory on the titan took less than 4 seconds. This means that even in the case the whole dataset does not fit into GPU memory, casting it into chunks and processing them one after the other by the GPU should not significantly impact the overall processing time.

VII. Extension to the DFG model

We believe that an extension of the DFG processing model of a processing framework such as AROM or Dryad to take into account GPU primitives can form a suitable platform for distributed heterogeneous computing. Such platform enables users to elegantly design data processing jobs which transparently make use of regular control-intensive CPU code and data parallel GPU-enabled code. In this section we will detail the required extensions to the DFG model of AROM and the implications on the rest of the framework to enable GPU related tasks.

The DFG model is a directed acyclic graph in which each node represents an operation and each edge represent a data dependency between operations. In a DFG processing framework each of these node will be scheduled for execution on an available core and each edge represents a data transfer.

With this representation, specific GPU kernel code can be encapsulated within an operator that the user can simply reuse and combine with other operators to form the final job. This permits to bundle operators which make use of the GPU for specific tasks into a library for the user. The idea is that the user will only handle code execution on the GPU through these operators and create a job by mixing regular operators, running CPU code, and these operators running parallel GPU code. Figure 4 depicts such a job. An additional advantage to this is that an operator can bundle higher-level primitives enforcing the best practices and patterns of GPU programming, for example reduce functions, sort, convolutions and transformations.

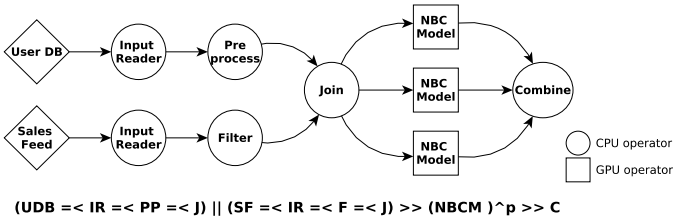

Fig. 4. DFG representation of a job composed of two data sources, preprocessing stages and a join operation whose results are fed to a GPU-enabled operators stage. Square nodes are operators executing code on the GPU. The Dyrad-like notation [2] of the job is also represented.

TABLE I. GPU kernel taxonomy with regards to their data transfer dependence, according to [18]

Kernel category	Description
Non-Dependent (ND)	The kernel does not depend on data transfer with the GPU, or the dependence is extremely small.
Dependent Streaming (DS)	The kernel is dependent on data transfer with the GPU but this is hidden by streaming transfer.
Single-Dependent-Host-to-Device (SDH2D)	Kernel depends on data transfer to GPU.
Single-Dependent-Device-to-Host (SDD2H)	Kernel depends on data transfer from the GPU.
Dual-Dependent (DD)	Kernel depends on data transfer from and to the GPU.

The scheduler of the framework would need to take into account the fact that some of the operators need to run code on GPU and map this to the GPU capability of the nodes in the cluster. A same *sorting* operation for example can be implemented in two different versions, one using the CPU and the other using the compute capabilities of the GPU. At runtime the scheduler will dynamically decide to run one version or the other depending on the capabilities of the available hosts. This will require to annotate an operator in the job graph to reflect the GPU-dependent nature of the operator, or define a new type of nodes in complement to the existing types.

The scheduler will also need to give higher priority to data locality in the case of GPU operators, as the data transfer to the GPU device can represent an additional overhead in the job. The scheduling of the operators should amortize this transfer time by scheduling the subsequent operators which make use of the data present in the GPU memory on the same physical node to eliminate the data transfer between GPUs of different nodes. Moreover, the scheduler could use hints on the data dependency of the operators in order to optimize their scheduling. For example [18] proposes a taxonomy for GPU jobs with regards to the dependency on memory transfer between the host and the GPU (Table I).

Finally, the DFG model has been known for being suitable to implement a higher-level query language on top of the framework. AROM and Stratosphere for example have a Pig interpreter which translates the Pig query execution plan into

a DFG job. Queries involving sorts, joins, merges, ... are translated into a DFG job and executed on a dataset. With regards to what we said above, parts of these queries can be executed on the GPU using the correct operators. This can lead a step closer to GPU-offloaded database processing such as described in [19] and [20].

VIII. Conclusion

In this paper we have presented a distributed processing framework suited for data mining whose computing nodes make use of the parallel processing power of GPUs to tackle data-parallel tasks. This framework falls under the context of DFG distributed processing frameworks such as AROM, Dryad and Stratosphere, to which its contribution is to show that on data-parallel jobs, performance can be gained by equipping the worker nodes with GPU compute enabled devices. We have highlighted the significant performance gain on two data-mining problems, the naive Bayes classifier and the k-means clustering, that we have engineered to run distributed and to use the parallel processing from the GPU. We believe that combining the best of both worlds, the distributed nature of the processing framework, and the individual parallel computing abilities of the nodes provided by the GPU, can add much value and form a suitable platform for distributed heterogeneous processing in the future.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Sixth Symposium on Operating System Design and Implementation OSDI'04*, 2004, pp. 1–13.

[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, 2007, pp. 59–72.

[3] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke, "Nephele / PACTs : A Programming Model and Execution Framework for Web-Scale Analytical Processing Categories and Subject Descriptors," in *Proceedings of the 1st ACM symposium on Cloud computing (SoCC'10)*, 2010.

[4] N.-L. Tran and S. Skhiri, "AROM: Processing Big Data with Data Flow Graphs and Functional Programming," in *Proceedings of the IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom)*, 2012.

[5] A. Alexandrov, S. Ewen, M. Heimel, F. Hueske, O. Kao, V. Markl, E. Nijkamp, and D. Warneke, "MapReduce and PACT-Comparing Data Parallel Programming Models." in *Proceedings of the 14th Conference on Database Systems for Business, Technology, and Web (BTW)*, 2011, pp. 25–44.

[6] S. Huang, J. Huang, Y. Liu, L. Yi, and J. Dai, "HiBench: A Representative and Comprehensive Hadoop Benchmark Suite," in *Proceedings of the ICDE Workshops 2010*, 2010.

[7] H. Xiao, "Towards Parallel and Distributed Computing in Large-Scale Data Mining: A Survey," Technical University of Munich, Tech. Rep., 2010.

[8] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Y. Yu, G. Bradski, A. Ng, and K. Olukotun, "Map-Reduce for Machine Learning on Multicore," *Advances in Neural Information Processing Systems*, vol. 19, pp. 281—-288, 2007.

[9] R. McDonald, K. Hall, and G. Mann, "Distributed Training Strategies for the Structured Perceptron," in *Proceedings of the 2010 Annual Conference of the North American Chapter of the ACL*, no. June, 2010, pp. 456–464.

[10] C. Böhm, R. Noll, and C. Plant, "Data Mining Using Graphics Processing Units," in *Transactions on Large-Scale Data- and Knowledge-Centered Systems I*, 2009, pp. 63–90.

[11] M. Benguigui and F. Baude, "Towards Parallel and Distributed Computing on GPU for American Basket Option Pricing," in *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*. Ieee, Dec. 2012, pp. 723–728.

[12] B. Hong-tao, H. Li-li, O. Dan-tong, L. Zhan-shan, and L. He, "K-Means on Commodity GPUs with CUDA," in *2009 WRI World Congress on Computer Science and Information Engineering*. Ieee, 2009, pp. 651–655.

[13] M. Zechner and M. Granitzer, "Accelerating K-Means on the Graphics Processor via CUDA," in *2009 First International Conference on Intensive Applications and Services*. Ieee, Apr. 2009, pp. 7–15.

[14] W. Fang, K. Lau, M. Lu, and X. Xiao, "Parallel Data Mining on Graphics Processors," Honk Kong University of Science and Technology, Tech. Rep., 2008.

[15] L. Lopes and B. Ribeiro, "GPUMLib: An Efficient Open-Source GPU Machine Learning Library," *International Journal of Computer Information Systems and Industrial Management Applications*, vol. 3, pp. 355–362, 2010.

[16] I. Dhillon and D. Modha, "A Data-Clustering Algorithm on Distributed Memory Multiprocessors," *Large-Scale Parallel Data Mining*, 2002.

[17] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, and M. Smelyanskiy, "Debunking the 100X GPU vs . CPU Myth : An Evaluation of Throughput Computing on CPU and GPU," in *Proceedings of the 37th annual international symposium on Computer architecture (ISCA '10)*, 2010, pp. 451–460.

[18] C. Gregg and K. Hazelwood, "Where is the Data? Why you Cannot Debate CPU vs. GPU Performance Without the Answer," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Ieee, Apr. 2011, pp. 134–144.

[19] P. Bakkum and K. Skadron, "Accelerating SQL Database Operations on a GPU with CUDA," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units - GPGPU '10*. New York, New York, USA: ACM Press, 2010, p. 94.

[20] P. Bakkum and S. Chakradhar, "Efficient Data Management for GPU Databases," NEC Laboratories America, Princeton, NJ, Tech. Rep.